Proof techniques for precision and progress of computational symbolic execution

Dries Vanoverberghe
Frank Piessens

Report CW 670, August 2014

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Proof techniques for precision and progress of computational symbolic execution

Dries Vanoverberghe
Frank Piessens

Report CW 670, August 2014

Department of Computer Science, KU Leuven

Abstract

Given a program and an assertion in that program, determining if the assertion can fail is one of the key applications of program analysis. Symbolic execution is a well-known technique for finding such assertion violations that can enjoy the following two interesting properties. First, symbolic execution can be precise: if it reports that an assertion can fail, then there is an execution of the program that will make the assertion fail. Second, it can be progressing: if there is an execution that makes the assertion fail, it will eventually be found. A symbolic execution algorithm that is both precise and progressing is a semi-decision procedure.

Recently, compositional symbolic execution has been proposed. It improves scalability by analyzing each execution path of each method only once. However, proving precision and progress is more challenging for these compositional algorithms. This paper investigates under what conditions a compositional algorithm is precise and progressing (and hence a semi-decision procedure).

Proof techniques for Precision and Progress of Compositional Symbolic Execution

Dries Vanoverberghe[1] and Frank Piessens[1]

[1]IBBT-DistriNet, KU Leuven, Celestijnenlaan 200a, 3001 Heverlee, Belgium.
email: Dries.Vanoverberghe@cs.kuleuven.be, Frank.Piessens@cs.kuleuven.be.

Abstract. Given a program and an assertion in that program, determining if the assertion can fail is one of the key applications of program analysis. Symbolic execution is a well-known technique for finding such assertion violations that can enjoy the following two interesting properties. First, symbolic execution can be precise: if it reports that an assertion can fail, then there is an execution of the program that will make the assertion fail. Second, it can be progressing: if there is an execution that makes the assertion fail, it will eventually be found. A symbolic execution algorithm that is both precise and progressing is a semi-decision procedure.

Recently, compositional symbolic execution has been proposed. It improves scalability by analyzing each execution path of each method only once. However, proving precision and progress is more challenging for these compositional algorithms. This paper investigates under what conditions a compositional algorithm is precise and progressing (and hence a semi-decision procedure).

Keywords: compositional; symbolic execution; precision; progress

1. Introduction

Given a program and an assertion in that program, determining whether the assertion can fail is one of the key applications of program analysis. There are two complementary approaches.

One can try to determine whether the assertion is *valid*, i.e. is satisfied in all executions of the program. This can be done using techniques such as type systems, abstract interpretation, or program verification. Such techniques are typically expected to be *sound*: if they report an assertion as valid, there will indeed be no execution that violates the assertion. However, these techniques suffer from false positives: they may fail to establish the validity of an assertion even if there is no execution that violates the assertion. In addition, they often require human effort to provide type annotations, loop invariants or method specifications.

Alternatively one can look for counterexamples by trying to determine inputs to the program that will make the assertion fail. One important technique for this approach is symbolic execution [Kin76], a well-known analysis technique to explore the execution traces of a program. The program is executed symbolically

using logical symbols for program inputs, and at each conditional the reachability of both branches is checked using an SMT solver. When reaching the assertion, the analysis determines if it can find values for the symbolic inputs that falsify the assertion. Such a technique can not prove the validity of an assertion, but it has the advantage of avoiding false positives (a property that we will call *precision*). Obviously, sound and precise approaches are complementary. This paper focuses on precise algorithms that require no additional human effort, and more specifically on precise symbolic execution.

The widespread use of symbolic execution both in research prototypes [TdH08, CGP+06, GLM08, NRTT09, CCC+05, BHK+07, APV07, MW07, PDEP08] as well as in industrial strength tools [CGP+06, GLM08] has shown that it is a viable tool to reduce the number of software defects. However, scalability remains one of the most important challenges. Recently, *compositional* symbolic execution [God07, AGT08] attempts to further improve the scalability of symbolic execution. With compositional symbolic execution, each execution path of a method is only analyzed once. The results of this analysis are stored in a so-called *summary* of the method, and are reused by all callers of the method.

Traditional whole-program (non-compositional) symbolic execution has two interesting properties that are not necessarily maintained in the compositional case. First, symbolic execution can be *precise*: if it reports that an assertion can fail, then there is an execution of the program that will make the assertion fail. Proving precision for whole-program symbolic execution is relatively easy: one has to prove that symbolic execution correctly abstracts concrete executions, and that the SMT solver is sound and complete (which it can be for the class of constraints it needs to solve).

Second, symbolic execution can *make progress* or be *progressing*: if there is an execution that makes the assertion fail, it will eventually be found. Therefore, there are no classes of programs where the analysis fails fundamentally. Again, making a symbolic execution algorithm progressing is relatively straightforward, for instance by making the algorithm explore the tree of possible paths through the program in a breadth-first manner. Since this tree is finitely-branching, a breadth-first exploration ensures that any node of the tree will eventually be visited. A symbolic execution algorithm that is both precise and progressing is a semi-decision procedure for the existence of counterexamples.

In some sense, symbolic execution is an algorithm to decide the satisfiability of certain reachability queries in the logic defined by the programming language. Precision is a soundness property and progress is a weakened completeness property of the satisfiability problem of the resulting logical system. However, the terms soundness and completeness are typically used in the context of the validity problem of logical systems, for example in verification systems or theorem provers. Therefore, we avoid the terms soundness and completeness on purpose to avoid confusion.

Although compositional symbolic execution is inspired by standard symbolic execution, the proofs of these important properties become much more challenging. In fact, some of the algorithms proposed recently are not necessarily semi-decision procedures. In this paper, we develop proof techniques for showing precision and progress of compositional symbolic execution algorithms.

More specifically, this paper makes the following contributions:

- We create a formal model of reachability analysis based on transition systems. This framework allows us to define fundamental properties such as soundness, completeness, precision and progress.
- We show how a symbolic execution algorithm for a small but powerful calculus can be integrated into this framework, and we demonstrate that this algorithm is precise and makes progress.
- We discuss how compositional symbolic execution differs from standard symbolic execution, and we motivate how this may affect the fundamental properties of the algorithm.
- We formally model the existing compositional symbolic execution, based on a small but powerful programming language.
- We show that any compositional symbolic execution algorithm based on this formal model is *precise*.
- We give sufficient conditions for an algorithm to be *progressing*, and therefore be a semi-decision procedure.

For the purpose of investigating precision and progress, the assertion in the program is not relevant. What matters is whether the symbolic execution algorithm correctly enumerates all the reachable program states. Hence, for the rest of this paper, we will consider symbolic execution algorithms to be algorithms that enumerate reachable program states. Such an algorithm is precise if any program state that it enumerates is also reachable by the program. It is progressing if any program state reachable by the program is eventually enumerated.

This paper has evolved from a paper which is published at the conference on Fundamental Approaches to Software Engineering 2011 [VP11]. In contrast with this prior work, this paper separates the formal model of reachability analysis from the algorithm for compositional symbolic execution. Next, this paper shows that the formal framework is suitable by creating a model for the symbolic execution algorithm and showing that this algorithm is precise and progressing. Finally, this paper includes more detailed versions of the proofs.

The rest of this paper is structured as follows. First, in Section 2 we show by means of examples that precision and progress are hard to achieve for compositional algorithms. Next, Section 3 introduces a formal model for reachability analysis algorithms and defines precision and progress. Then we introduce a small but powerful programming language in Section 4. Next, we show how symbolic execution can be modeled using this framework (Section 5). Section 6 presents compositional symbolic execution and creates a formal model of it based on transition systems. Next, we show that this algorithm is precise and progressing (Section 7). Finally, we discuss related work in Section 8 and conclude in Section 9.

2. Motivation

Traditional symbolic execution [Kin76] explores paths through the program by case splitting whenever the execution reaches a branch. Since loops are also just branches that are encountered multiple times, this implies that loops are lazily unrolled, potentially an infinite number of times [1]. When a method call is reached, the target method is symbolically executed using the given arguments. Therefore, if the program calls a given method several times, the execution paths in that method will be re-analyzed for each call. The key idea of compositional algorithms is to avoid this repeated analysis. Instead, execution paths are explored for each method independently. The results of this exploration are stored in a *method summary*. Method calls are no longer inlined: the result of a method call is computed based on the current summary of the target method and therefore each method call is analyzed in one single step. Compositional symbolic execution has been shown [God07, AGT08, GNRT10] to improve performance, but maintaining precision and progress is challenging.

2.1. Precision

Compositional symbolic execution creates two potential causes of imprecision. First, when there is insufficient information about the calling context of a method, then one might conclude that unreachable program locations are reachable. For example, the highlighted statement in the method *P2* in Figure 1 is unreachable in the current program because the method *P1* only calls *P2* with argument $x \mathrel{!}= 0$. However, if one would analyze *P2* independently of *P1*, the analysis might conclude that the highlighted statement is reachable. In other words, since reachability is a whole-program property, we need to maintain some whole-program state even in a compositional analysis. The example algorithm we discuss later will do so by maintaining an invocation graph.

Second, when a method returns and the analysis loses information about the relation between the arguments of the method and the return value, then the analysis might incorrectly conclude that a program location is reachable. For example, the highlighted statement in the method *P1* in Figure 1 is unreachable. When the analysis over-approximates the result of *P2* by the relation $result == 0 \vee result == 1 \vee result == -1$, then the highlighted location is reachable. To maintain precision, method summaries should not introduce such approximations.

2.2. Progress

Non-compositional symbolic execution builds one global execution tree where leaf nodes represent either final program states, unreachable program states, or program states that require further analysis. Given a fair strategy to select such leaf nodes for further analysis, it is easy to show that the depth of the highest

[1] Developer provided or automatically synthesized invariants can be used to create sound analyzers for particular classes of programs. This paper considers the general case where such invariants are not present or cannot be inferred

```
int P1(int x) {
  if(x != 0){
    int r2 = P2(x);
    if(x > 0 && r2 != 1) return −1;
  }
  return 0;
}
```

```
int P2(int u) {
  if(u == 0) return 0;
  else if(u > 0) return 1;
  else return − 1;
}
```

Fig. 1. Example program for precision

```
int M1(int x) {
  while (x > 0) {
    int y = M2(x, x);
    if(y < 0) return −1;
    x−−;
  }
  return 0;
}
```

```
int M2(int u, int v) {
  int w = 0;
  while (u > 0) {
    if(v <= 0) return w;
    u−−; v−−; w++;
  }
  return w;
}
```

Fig. 2. Example program for progress

unexplored node keeps increasing and hence that any finite execution path will eventually be completely analyzed. This implies progress for non-compositional symbolic execution.

For compositional symbolic execution, the situation is more complex due to two reasons. First, as we discussed above, in order for method summaries to be precise, they must depend on the calling context. Hence, the discovery of a new call site for a method may increase the number of reachable points in the method and unreachable leaf nodes need to be reanalyzed taking into account the new calling context.

Secondly, when analyzing a method call, a compositional analysis relies on the summary of the target method for computing the return value. However, method summaries change over time when the analysis discovers new returns. As a consequence, nodes that were deemed unreachable based on the summary of the method must be reanalyzed when that method summary is updated.

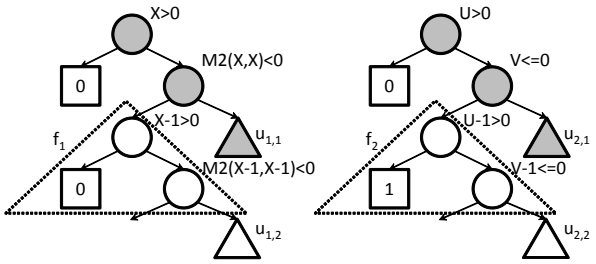
The progress argument for non-compositional symbolic execution relies essentially on the fact that unreachable leaf nodes remain unreachable for the rest of the analysis. With compositional symbolic execution, this premise is no longer satisfied. Furthermore, it is impossible to guarantee that any finite execution path within the execution tree of a single method will eventually be completely analyzed. The program in Figure 2 provides an example of this phenomenon.

First, we explain the program: The two highlighted statements are both unreachable, and therefore the method *M1* returns 0 for any input. To understand this, two invariants are important: First, the method *M1* only calls the method *M2* with parameters $u = v$ with $u > 0$. Second, if the parameters u and v of *M2* are greater than zero, then *M2* returns the minimum of u and v.

Figures 3a and 3b show the execution trees of *M1* and *M2*. Each circle represents a case split in the program, and the corresponding condition is written in the upper-right corner. From a circle, the arc to the left (right) means that the condition is false (true). Squares are final nodes, and imply that the method returns with the return value written inside the square. Triangle denotes unreachable nodes.

Let $u_{i,j}$ be the analysis step that checks the j-th unreachable node of *Mi*, and f_i the sequence of analysis steps that explores the reachable part of the execution tree of *Mi*. The sequence f_1 causes a new invocation from *M1* to *M2* and therefore resets the unreachable nodes in *M2*. The sequence f_2 causes a new return and therefore resets the unreachable nodes in *M1*. Let $u_{i,2..}$ be the sequence $u_{i,2}, \ldots, u_{i,n}$ where $u_{i,n}$ checks the deepest unreachable node of *Mi*. Suppose we analyze according to the fair schedule $f_1, f_2, [u_{2,1}, f_1, u_{2,2..}, u_{1,1}, f_2, u_{1,2..}]^*$. Then the grey nodes in the execution trees, that are only at depth 3 in the tree will be of status needs-further-analysis an infinite number of times. Hence, the depth of the analysis never stays larger than 3, and the given schedule is a counter-example for the traditional proof of progress.

This shows that progress for compositional symbolic execution can not be proved by mimicking the proof



(a) Execution tree of *M1* (b) Execution tree of *M2*

Fig. 3. Execution trees for *M1* and *M2*

for the non-compositional case on a per-method basis. In Section 7, we will propose an alternative technique to prove progress for compositional symbolic execution.

3. The reachability problem

Parameterized unit tests [TS05] are test methods where parameters can be used to specify that the test scenario should work for all input values. Given such a test scenario with parameters, deciding whether the execution can reach an erroneous state is one of the key approaches to automate the test process. In this paper, we abstract from the details of the test case generation algorithm or the language to specify the test oracle. We focus exclusively on the reachability problem. We create a formal definition for reachability analysis algorithms and specify some of the key properties which are important in the context of test automation.

Alternatively, testing can be modeled as property checking: given pre- and postconditions for a piece of code, search for an input that satisfies the precondition but invalidates the postcondition. In principle, it is possible to transform a property checking problem into a reachability problem provided that the control flow of the programming language is expressive enough to model the pre-and postconditions. Similarly, the converse is also possible. Therefore, they are equally suitable. By modeling the reachability problem, we avoid the need to specify a separate assertion language. Furthermore, this most closely relates to the situation in practice, where testers use the same language as the program to encode the properties.

First, we give an explicit definition of the reachability problem. To do so, we assume the availability of a programming language which defines the syntax of programs, the execution states and a small-step operational semantics $s \rightarrow^* s'$ (Section 4 will introduce these notions). Based on this, reachability can be defined as follows:

Definition 1 (Reachability). A state s' is reachable from a state s if and only if $s \rightarrow^* s'$. A state s is reachable in a program pr (denoted as $\models reach(pr, s)$) if and only if s is reachable from the initial state s^0_{pr} of the program.

To analyze the fundamental properties of a reachability analysis algorithm, it is convenient to model an algorithm as a transition system $a \Rightarrow a'$ (and $\Rightarrow^*$ its reflexive transitive closure) which starts in an initial analysis state a^0. Non-terminating runs of the algorithm can be truncated after any number of transitions. In addition, the predicate $\vdash^a reach(p, s)$ denotes that the analysis concludes the reachability of the state s in the program p in an analysis state a.

Such a transition system is precise if and only if the conclusion in any reachable analysis state is sound. This means that whenever a state is concluded reachable in an analysis state, it is truly reachable in the programming language:

Definition 2 (Precision). For each program pr, concrete state s, and analysis state a such that $a^0_{pr} \Rightarrow^* a$, $\vdash^a reach(pr, s)$ implies $\models reach(pr, s)$.

Precision is one of the most important properties in the context of test automation, and it is arguably equally important as the soundness of a type system or program verifier. Precision guarantees that whenever the algorithm discovers a software defect, it is truly a defect of the actual system.

The converse property, completeness means that any state which is reachable in the programming language must also be concluded reachable in all reachable analysis states:

Definition 3 (Always Complete). For each program pr, concrete state s, and analysis state a such that $a^0_{pr} \Rightarrow^* a$, $\models reach(pr, s)$ implies $\vdash^a reach(pr, s)$.

Due to undecidability, a precise reachability analysis algorithm can not be complete in all reachable analysis states. However, these algorithms can incrementally discover more and more reachable states. This incremental nature is captured by monotonicity:

Definition 4 (Monotonicity). For each program pr, concrete state s, and analysis states a, a' such that $a \Rightarrow a'$, if $\vdash^a reach(pr, s)$ then $\vdash^{a'} reach(pr, s)$.

For a monotonic analysis, progress is the next best thing with respect to completeness: for any reachable concrete state, eventually there is an analysis state that concludes reachability for that concrete state:

Definition 5 (Progress). For each program pr and each concrete state s, if $\models reach(pr, s)$ then for all analysis states a' such that $a^0_{pr} \Rightarrow^* a'$ there exists an analysis state a such that $a' \Rightarrow^* a$ and $\vdash^a reach(pr, s)$. In other words, for each reachable concrete state s, there always eventually is an analysis state that concludes s is reachable.

When an analysis is precise, monotonic and progressing, it is a semi-decision procedure.

4. Programming language

In this section, we introduce a small intermediate language that is particularly well-suited for presenting compositional symbolic execution. It only retains the structure of the program that is essential: the structure of the control flow graph per procedure, and the calls and returns between procedures. The language focuses on sequential programs. Besides this restriction, all relevant more complicated language features can be translated to this core (e.g. parameters, return values or loops, ...). For brevity, we also assume that the program does not contain (mutually) recursive methods. Although the algorithm in Section 6 depends on this simplification, our prototype implementation supports recursion by inferring ranks (See Section 6.5).

A program p is a tuple $\langle M_p, G_p, m^0_p \rangle$ where M_p is a set of methods, G_p is a set of global variables and $m^0_p \in M_p$ is a distinguished entry method. A method definition m for the program p is a tuple $\langle L_m, N_m, \lambda_m, n^0_m \rangle$ where L_m is a finite set of local variables disjoint from the global variables G_p, N_m is a finite set of program locations, $n^0_m \in N_m$ is a distinguished entry location and $\lambda_m : N_m \to Commands_{m,p}$ maps each program location to a command. The sets of local variables and program locations of different methods are disjoint.

A command c for the method m of the program p is either:

- An assignment **assign** x, e, n where $x \in L_m \cup G_p$, e is a side-effect free expression over $L_m \cup G_p$ and constants, and $n \in N_m$ is a program location. This command updates the value of the variable x, and continues in location n.
- A conditional **if** e, n_t, n_f where e is a side-effect free expression over $L_m \cup G_p$, and $n_t, n_f \in N_m$ are program locations. If the expression e evaluates to true (false) the execution continues in location n_t (n_f).
- A call **call** m_t, n where $m_t \in M_p$ is the target method and $n \in N_m$ is a program location. This command invokes the method m_t and continues in location n.
- A return **ret** returns from the current method.

For each variable v, $\mathcal{D}(v)$ represents the value domain of the variable. A valuation σ_V is a partial function that maps each variable $v \in V$ to a value $val \in \mathcal{D}(v)$. Each domain has a default element $\mathcal{D}_0(v)$, and the default valuation σ^d_V for a set of variables V maps each variable $v \in V$ to $\mathcal{D}_0(v)$.

An execution state $s \in S_p$ for the program p is a tuple $\langle \sigma_G, \overline{f} \rangle$ where

- σ_G is the current valuation for G_p
- $\overline{f} \in F^*_p$ is a sequence of frames for p (sequences are either empty (nil) or a concatenation $h; \overline{t'}$ of a head h and a tail $\overline{t'}$)

$$\frac{\lambda_m(n) = \textbf{assign}\ x, e, n' \qquad \textbf{let}\ \sigma_T := (\sigma_G \cup \sigma_L) \oplus x \mapsto eval(\sigma_G \cup \sigma_L, e)}{\langle \sigma_G, \langle m, n, \sigma_L \rangle; \overline{f} \rangle \to \langle \sigma_T|_{G_p}, \langle m, n', \sigma_T|_{L_m} \rangle; \overline{f} \rangle}\ \text{ASSIGN}$$

$$\frac{\lambda_m(n) = \textbf{if}\ e, n_t, n_f \qquad eval(\sigma_G \cup \sigma_L, e) = true}{\langle \sigma_G, \langle m, n, \sigma_L \rangle; \overline{f} \rangle \to \langle \sigma_G, \langle m, n_t, \sigma_L \rangle; \overline{f} \rangle}\ \text{COND-T} \qquad \frac{\lambda_m(n) = \textbf{if}\ e, n_t, n_f \qquad eval(\sigma_G \cup \sigma_L, e) = false}{\langle \sigma_G, \langle m, n, \sigma_L \rangle; \overline{f} \rangle \to \langle \sigma_G, \langle m, n_f, \sigma_L \rangle; \overline{f} \rangle}\ \text{COND-F}$$

$$\frac{\lambda_m(n) = \textbf{call}\ m_t, n'}{\langle \sigma_G, \langle m, n, \sigma_L \rangle; \overline{f} \rangle \to \langle \sigma_G, f^0_{m_t}; \langle m, n', \sigma_L \rangle; \overline{f} \rangle}\ \text{CALL} \qquad \frac{\lambda_m(n) = \textbf{ret} \qquad \overline{f} \neq nil}{\langle \sigma_G, \langle m, n, \sigma_L \rangle; \overline{f} \rangle \to \langle \sigma_G, \overline{f} \rangle}\ \text{RET}$$

Fig. 4. Operational semantics

A frame $f \in F_p$ for the program p is a tuple $\langle m, n_m, \sigma_{L_m} \rangle$ where

- $m \in M_p$ is the current method.
- $n_m \in N_m$ is the current program location.
- σ_{L_m} is the current valuation for L_m

Figure 4 defines the operational semantics $\to \subseteq S \times S$, which gives an interpretation to the commands, and $\to^*$ is its reflexive transitive closure. The premises **let** $x := y$ are not real conditions, they provide abbreviations for long expressions. We depend on partial functions: Let $Dom(f)$ be the domain of the partial function f. The union of two partial functions f_1 and f_2 with disjoint domains is denoted by $f_1 \cup f_2$. Restriction of a partial function f to the domain D is denoted by $f|_D$. A singleton partial function $x \mapsto r$ maps the input x to a result r. The overriding $f_1 \oplus f_2$ of a partial function f_1 by a partial function f_2 is the disjoint union of $f_1|_{Dom(f_1) \setminus Dom(f_2)}$ and f_2. In addition, the evaluation $eval(\sigma_V, e)$ of an expression e in a valuation σ_V is defined by substituting each variable $v \in V$ by $\sigma_V(v)$ in e in the usual way.

The execution of a program p starts in the initial state $s^0_p = \langle \sigma^0_{G_p}, f^0_{m^0_p} \rangle$ with $f^0_{m^0_p}$ the initial frame for m^0_p, and $\sigma^0_{G_p}$ the input valuation for the global variables G_p. The initial frame for a method m is $f^0_m = \langle m, n^0_m, \sigma^d_{L_m} \rangle$.

A state s is reachable from a state s' if and only if $s \to^* s'$. A state s is reachable in a program pr (denoted as $\models reach(pr, s)$) if and only if s is reachable from the initial state s^0_{pr} of the program.

5. Symbolic execution

In this section, we create a formal model for the essence of symbolic execution in the framework of Section 3 and we discuss its fundamental properties.

5.1. Overview

The analysis state maintains a prefix of the global execution tree of the program, which is defined as a set of leaf nodes. Each leaf node $\langle \nu, pc \rangle$ contains:

- A symbolic execution state ν, and
- A path condition pc.

The path condition defines the inputs (i.e. the values of the global variables) that will drive the execution of the method along this path. The symbolic execution state represents the state of execution after executing the path. Symbolic execution states are defined like concrete execution states, except that all valuations are symbolic valuations i.e. any variable has a symbolic expression instead of a concrete value.

In addition, the analysis tracks all reachable program states it has enumerated. For this purpose, the analysis state contains a set of leaf nodes that succeeded the reachability check for each method. Based on

```
AnalysisState Step(AnalysisState a) {
  ϵ = Choose(a);
  if(Check(ϵ.pc)) {
    rs′ = a.rs ∪ {ϵ};
    π = SyInt(ϵ);
    tree′ = a.tree \ {ϵ} ∪ π;
    return new AnalysisState(tree′, rs′)
  } else {
    tree′ = a.tree \ {ϵ};
    return new AnalysisState(tree′, a.rs);
  }
}
```

Fig. 5. High level algorithm of traditional symbolic execution

this information, reachability conclusion is defined: A state s is concluded reachable in an analysis state a (denoted $\vdash^a reach(pr, s)$) if and only if the set of reachable states of a contains a leaf node $\langle \nu, pc \rangle$, and s is a concretization of ν where the global variables satisfy the path condition pc.

Usually, one is only interested whether a point in a program is reachable (e.g. a location n in a method m). Therefore, implementations often store reachable program points instead of leaf nodes or avoid the reachability set completely by reporting an error when reaching a distinguished error-location. However, reachability conclusion of arbitrary states is essential for inductive invariants that enable the precision and progress proofs.

Finally, an analysis state can be defined as a tuple $\langle tree, rs \rangle$ where

- $tree$ is the execution tree of the program.
- rs is a set of reachable leaf nodes.

In the initial analysis state a_p^0, the set of reachable leaf nodes is empty, and the execution tree starts with a symbolic execution state at the entry of the main method, where the value of all global variables contains a fresh symbol.

The high level overview of one analysis step[2] of the symbolic execution algorithm is shown in Figure 5. During each step, the algorithm chooses a leaf node $\epsilon \in tree$ in the current prefix of the execution tree. Then, the algorithm checks whether there exists an input $\sigma^0_{G_p}$ which satisfies the path condition pc_ϵ.

If there is no such input, the leaf node ϵ can be removed from the execution tree. Otherwise, ϵ is added to the set of reachable leaf nodes rs, and symbolic execution continues with the interpretation ($SyInt(\epsilon)$) of the symbolic state ν_ϵ of ϵ. When symbolic interpretation finishes, it returns a set of new leaf nodes π, and the current leaf node ϵ is replaced by the new leaf nodes. All method calls in this algorithm are guaranteed to terminate, and therefore one step of the algorithm always terminates.

We now zoom in on some aspects of the algorithm that are of importance for precision and progress.

5.2. Symbolic interpretation

During symbolic interpretation, the algorithm applies one or more execution steps to the symbolic state. Figure 6 defines the symbolic interpretation rules for the language of Section 4. The rules of symbolic interpretation have the following structure:

$$\frac{premises}{\epsilon \Rightarrow \pi}$$

When a rule is applied to a leaf node ϵ, the rule transforms this leaf node into a set of new leaf nodes. In practice, most rules result into a single leaf node with identical path condition. In this case, the algorithm

[2] One analysis step performs multiple symbolic interpretation steps when no branches are encountered.

$$\frac{\lambda_m(n) = \textbf{assign}\ x, e, n' \qquad \textbf{let}\ \varsigma_T := (\varsigma_G \cup \varsigma_L) \oplus x \mapsto seval(\varsigma_G \cup \varsigma_L, e)}{\langle\langle \varsigma_G, \langle m, n, \varsigma_L\rangle; \overline{\varrho}\rangle, pc\rangle \Rightarrow \{\langle\langle \varsigma_T|_{G_p}, \langle m, n', \varsigma_T|_{L_m}\rangle; \overline{\varrho}\rangle, pc\rangle\}}\ \textsc{assign}$$

$$\frac{\lambda_m(n) = \textbf{if}\ e, n_t, n_f \qquad \textbf{let}\ b := seval(\varsigma_G \cup \varsigma_L, e) \qquad \textbf{let}\ \epsilon_1 := \langle\langle \varsigma_G, \langle m, n_t, \varsigma_L\rangle; \overline{\varrho}\rangle, pc \wedge b\rangle \qquad \textbf{let}\ \epsilon_2 := \langle\langle \varsigma_G, \langle m, n_f, \varsigma_L\rangle; \overline{\varrho}\rangle, pc \wedge \neg b\rangle}{\langle\langle \varsigma_G, \langle m, n, \varsigma_L\rangle; \overline{\varrho}\rangle, pc\rangle \Rightarrow \{\epsilon_1, \epsilon_2\}}\ \textsc{cond}$$

$$\frac{\lambda_m(n) = \textbf{call}\ m_t, n'}{\langle\langle \varsigma_G, \langle m, n, \varsigma_L\rangle; \overline{\varrho}\rangle, pc\rangle \rightarrow \{\langle\langle \varsigma_G, \varrho^0_{m_t}; \langle m, n', \varsigma_L\rangle; \overline{\varrho}\rangle, pc\rangle\}}\ \textsc{call} \qquad \frac{\lambda_m(n) = \textbf{ret} \qquad \overline{\varrho} \neq nil}{\langle\langle \varsigma_G, \langle m, n, \varsigma_L\rangle; \overline{\varrho}\rangle, pc\rangle \rightarrow \{\langle\langle \varsigma_G, \overline{\varrho}\rangle, pc\rangle\}}\ \textsc{ret}$$

Fig. 6. Traditional symbolic interpretation rules

can continue with the next step of symbolic interpretation without re-checking the path condition. Therefore, the method $SyInt$ used in Figure 5 can be defined as repeated symbolic interpretation:

$$\frac{\epsilon \Rightarrow \pi}{\epsilon \Rightarrow^+ \pi} \qquad\qquad \frac{\epsilon \Rightarrow \{\epsilon'\} \qquad \epsilon' \Rightarrow^+ \pi}{\epsilon \Rightarrow^+ \pi}$$

The actual symbolic interpretation rules in Figure 6 are standard. The rules for ASSIGN, CALL and RETURN rule are similar as the corresponding rules for concrete execution. They take a leaf node, transform its symbolic state and return a set containing the transformed leaf node. The rule for COND on the other hand groups the concrete rules. It returns a set containing two leaf nodes, one for each potential outcome of the conditional.

5.3. Reachability checking

Finally, to check reachability ($Check(pc)$), the algorithm uses an SMT-solver to check the satisfiability of the path condition. All integers and integer operations are precisely encoded in the theory of bitvectors and memory operations are encoded using the theory of arrays. Therefore, all constraints are decidable, and the underlying SMT-solver is sound and complete for the subset of constraints that is generated.

5.4. Properties

The key property to show precision is that the prefix of the execution is always precise, i.e. if an input satisfies the path condition of a leaf node, then the execution starting with this input eventually reaches the concretization of the symbolic state of that leaf node. This can easily be proved by induction on $\Rightarrow$ since the symbolic interpretation rules are structurally identical to the operational semantics of the language.

Before adding any leaf node to the set of reachable leaf nodes, an SMT-solver checks whether its path condition is satisfiable. Therefore, if the SMT-solver is sound (as a satisfiability checker) then traditional symbolic execution is precise.

Next, symbolic execution is clearly monotonic, since it never removes any of the leaf nodes from the set of reachable leaf nodes.

Finally, to show progress, it suffices to show that symbolic execution always eventually increases the depth of the prefix of the execution tree provided that the search strategy is fair, i.e. any leaf node is guaranteed to be chosen eventually. As long as the SMT-solver is complete (as a satisfiability checker), the algorithm will perform symbolic interpretation, which will increase the depth. Therefore, traditional symbolic execution is progressing as long as the search strategy is fair.

6. Compositional symbolic execution

In this section, we model the essence of existing compositional symbolic execution algorithms in order to formally study their precision and progress properties.

The difference between compositional [God07, AGT08] and traditional symbolic execution is in the treatment of method calls and returns. In traditional symbolic execution, a call adds a new symbolic frame for the target method and continues execution until a return command pops this frame. Therefore, when a method is called twice, a path through the method is computed twice, even if it is guaranteed to follow the same path. Compositional symbolic execution explores the execution trees of each method in isolation. This results in a partition for each method (also called the summary). When a call is encountered, compositional symbolic execution uses the summary of the target method to compute the effect on the symbolic state.

6.1. Overview

The analysis state maintains a summary per method, which is a set of leaf nodes of the current prefix of the execution tree of the method. A leaf node $\langle stat, \nu, pc \rangle$ contains:

- A status $stat$, which is either unknown, finished or unreachable,
- A symbolic execution state ν,
- A path condition pc.

The path condition defines the inputs (i.e. the values of the global variables) that will drive the execution of the method along this path. The symbolic execution state represents the state of execution after executing the path. The status indicates whether (a) the path is a complete path through the method, i.e. the method returns after this path (finished status) (b) the path is unreachable (unreachable status) (c) further exploration of continuations of this path are needed (unknown status). Symbolic execution states are defined like concrete execution states, except that all valuations are symbolic valuations i.e. any variable has a symbolic expression instead of a concrete value.

The summaries only maintain per-method information. As we have shown in Section 2, it is necessary to maintain some whole program information in order to be precise. In particular, it is important to precisely track reachable method invocations and returns.

Initially, only the main method is reachable. As the analysis progresses, any call that is discovered is stored in an invocation graph. This graph is represented as a set of invocations, where each invocation is a tuple $\langle m_s, m_t, \varsigma_G, pc \rangle$. The methods m_s and m_t are the source and target methods, and ς_G and pc are the symbolic values of the global variables and the path condition at the moment of the invocation. Reachability checking will use the information in the call graph to decide whole-program reachability.

To support discovery of new returns efficiently, the return values of method calls can be modeled using logic function symbols [AGT08]. This is a common approach which has also been used in the Key Tool[ABB+05]. The symbolic execution of a call is defined in terms of these function symbols. The interpretations of the function symbols are constructed using the current summary. As the analysis progresses, they become precise for more and more inputs. We discuss this in more detail in Section 6.2.

In addition, the analysis tracks all reachable program states it has enumerated. For this purpose, the analysis state contains a set of leaf nodes that succeeded the reachability check for each method. Based on this information, reachability conclusion is defined. If a leaf node $\langle stat, \nu, pc \rangle$ is in the reachable set of the method m in a given analysis state a, then any concretization of its symbolic state ν with global variables satisfying pc is concluded reachable in m. A state s is concluded reachable in an analysis state a (denoted $\vdash^a reach(pr, s)$) if and only if either

- s is concluded reachable in the entry method m^0_{pr} in a or
- there is a state s' such that $\vdash^a reach(pr, s')$ and s' calls m and s is concluded reachable in m.

As with traditional symbolic execution, implementations often need not store the reachability set. Instead, they can focus on distinguished error nodes or report reachable locations only. However, reachability conclusion of arbitrary states is essential for inductive invariants that enable the precision and progress proofs.

Finally, an analysis state can be defined as a tuple $\langle sum, invs, rs \rangle$ where

- sum is a function that maps each method m to a set of leaf nodes (its summary).

$$
\begin{array}{lcl}
a & ::= & \langle sum, invs, rs \rangle \\
i & ::= & \langle m_s, m_t, \varsigma_G, pc \rangle \\
\epsilon & ::= & \langle stat, \nu, pc \rangle \\
\varrho & ::= & \langle m, n, \varsigma_{L_m} \rangle \\
\nu & ::= & \langle \varsigma_{G_p}, \varrho \rangle
\end{array}
$$

Fig. 7. Definition of analysis states

$$
\begin{array}{lcl}
a_p^0 & ::= & \langle sum_p^0, \emptyset, rs_p^0 \rangle \\
sum_p^0 & ::= & \bigcup_{m \in M_p} m \mapsto \{\epsilon_m^0\} \\
rs_p^0 & ::= & \bigcup_{m \in M_p} m \mapsto \emptyset \\
\epsilon_m^0 & ::= & \langle unk, \nu_m^0, true \rangle \\
\nu_p^0 & ::= & \langle \varsigma_{G_p}^0, \varrho_{m_p^0}^0 \rangle \\
\varrho_m^0 & ::= & \langle m, n_m^0, \sigma_{L_m}^d \rangle
\end{array}
$$

Fig. 8. Initial analysis state

- $invs$ is a set of invocations.
- rs is a function that maps each method to a set of reachable leaf nodes.

Figure 7 summarizes all definitions with respect to analysis states.

In the initial analysis state a_p^0 (See Figure 8), the invocation graph and the sets of reachable leaf nodes are empty. Each summary starts with a symbolic execution state at the entry of the method, where the value of all global variables contains a new symbol.

The high level overview of one step of the compositional symbolic execution algorithm is shown in Figure 9. During each step, the algorithm chooses a method m and a leaf node $\epsilon \in sum_a(m)$ with unknown status. Then, the algorithm checks whether there exists an input $\sigma_{G_p}^0$ such that the execution enters the method m and the global variables satisfy the path condition pc_ϵ of ϵ ($Check(a, m, \epsilon.pc)$). If there is no such input, the status of the ϵ is changed to unreachable. Otherwise, ϵ is added to the set of reachable leaf nodes of m, and symbolic execution continues with the interpretation ($SyInt(a, m, \epsilon)$) of the symbolic state ν_ϵ of ϵ. When symbolic interpretation finishes, it returns a set of new leaf nodes, and the current leaf node ϵ is replaced by the new leaf nodes ($ReplaceLeaf$). All method calls in this algorithm are guaranteed to terminate, and therefore one step of the algorithm always terminates.

We now zoom in on some aspects of the algorithm that are of importance for precision and progress.

6.2. Symbolic interpretation

Figure 13 shows the rules for symbolic interpretation ($SyInt$).

As pointed out in the previous section, the analysis uses uninterpreted function symbols to support discovery of new returns as the analysis progresses. The algorithm models the effect of the method m on

```
AnalysisState Step(AnalysisState a) {
  (m, ε) = Choose(a);
  if(Check(a, m, ε.pc)) {
    a' = AddReachable(a, m, ε);
    (a'', π) = SyInt(a', m, ε);
    return ReplaceLeaf(a'', m, ε, π);
  } else {
    return MarkUnreach(a, m, ε);
  }
}
```

Fig. 9. High level algorithm of compositional symbolic execution

$$interps(sum) = \bigcup_{m \in M_p} interp(m, \{\epsilon | \epsilon \in sum(m), stat_\epsilon = fin\})$$
$$interp(m, \pi) = rc_m \mapsto interp_{rc}(m, \pi) \cup (\bigcup_{v \in G_p} rv_{m,v} \mapsto interp_{rv}(m, v, \pi))$$
$$interp_{rc}(m, \pi) = \bigvee_{\langle fin, \nu, pc \rangle \in \pi} pc$$
$$interp_{rv}(m, v, \emptyset) = \mathcal{D}_0(v)$$
$$interp_{rv}(m, v, \langle fin, \nu, pc \rangle \cup \pi) = ite\ pc\ \varsigma_{G_{\nu(v)}}\ interp_{rv}(m, v, \pi)$$

Fig. 10. Interpretation of uninterpreted function symbols

the global variable v as an uninterpreted function symbol $rv_{m,v}$. This uninterpreted function symbol takes a value for each global variable as input and returns the updated value. When a method m is called with global variables ς_{G_p}, then the function application $rv_{m,v}(\varsigma_{G_p})$ models the value for the global variable v after executing m.

In addition, the method summaries are *partial*: there is no information about unexplored paths through a method. To deal with this, the algorithm models the set of global variable valuations that follow a finished path using an uninterpreted predicate rc_m which takes the value of the global variables as arguments.

During reachability checking, the algorithm computes the interpretation for the uninterpreted symbols using the method summaries (Figure 10) and replaces them using substitution (e.g. $int(a, pc)$ replaces all occurrences of the uninterpreted symbols by their interpretation).

For precision, it is essential that the interpretation of the uninterpreted symbols is precise: Whenever the return condition $rc_m(\sigma_{G_p})$ is true, the execution of the method m starting with global variables σ_{G_p} eventually reaches a return command, and each global variable v must equal $rv_{m,v}(\sigma_{G_p})$.

Definition 6 (Precision of interpretations). For each program p, each analysis state a, the interpretations $interps(sum_a)$ are precise if and only if for each method $m \in M_p$ and each global input valuation σ_{G_p} that satisfies $int(a, rc_m(\sigma_{G_p}))$, $\langle \sigma_{G_p}, f^0_m \rangle \rightarrow^* \langle \sigma'_{G_p}, \langle m, n, \sigma_L \rangle; nil \rangle$ where $\lambda_m(n) = \textbf{ret}$ and $\sigma'_{G_p}(v) = int(a, rc_{m,v}(\sigma_{G_p}))$ for any variable $v \in G_p$.

Definition 7 (All reachable). In an analysis state a, all concrete states on the execution run $s \rightarrow^* s'$ are reachable (denoted $allReach(a, s \rightarrow^* s')$) if and only if $\vdash^a reach(pr, s)$ and $s = s'$ or $s \rightarrow s''$ and $allReach(a, s'' \rightarrow^* s')$.

Definition 8 (Restricted completeness of interpretations). For each program p, each analysis state a, the interpretations $interps(sum_a)$ are restricted complete if and only if for all states $s = \langle \sigma_{G_p}, f^0_m \rangle$ and $s' = \langle \sigma'_{G_p}, \langle m, n, \sigma_L \rangle; nil \rangle$ such that $s \rightarrow^* s'$, $\lambda_m(n) = \textbf{ret}$ and $allReach(a, s \rightarrow^* s')$, the return condition $int(a, rc_m(\sigma_{G_p}))$ is satisfied and $\sigma'_{G_p}(v) = int(a, rc_{m,v}(\sigma_{G_p}))$ for any variable $v \in G_p$.

The treatment of assignment and branches is similar to the treatment in non-compositional symbolic execution: For an assignment, symbolic interpretation performs the same operation but on symbolic expressions instead of concrete values. For branches, symbolic interpretation creates a new leaf node for each branch and conjoins the branch condition or its negation to the path condition.

The rule CALL in Figure 13 creates a new leaf node where the return condition is added to the path condition, and the return values are used to update the global variables. As mentioned in Section 2, some leaf nodes can become reachable by performing a call, and progress requires that all such leaf nodes are reconsidered. The algorithm conservatively reconsiders all unreachable leaf nodes of methods that are transitively reachable in the invocation graph by marking them as unknown (using the function $rec(a, m)$, defined more precisely in Section A). In practice, more intelligent re-evaluation strategies can take the context into account in order to minimize the number of affected nodes, but this is beyond the scope of the formal model.

The RETURN rule (See Figure 13) marks the unknown leaf node as finished, and thereby the interpretations of the current method change. In addition, the RETURN rule marks all unreachable leaf nodes that depend on the return condition as unknown again (using the function $rer(a, m)$, also defined in Section A). This is again essential to maintain progress.

For precision, the symbolic interpretation algorithm must maintain precision of the leaf nodes, i.e. if an input is a member of a leaf node, then the execution starting with that input eventually reaches the concretization of the symbolic state (the symbolic state after replacing the input symbols with the concrete

input). In addition, all invocations $\langle m_s, m_t, \varsigma_G, pc\rangle$ in the invocation graph must be precise: If an input satisfies the condition pc, then the execution of m_s starting with that input reaches a call to the method m_t and the global variables are the concretization of ς_G.

In the following definition, we assume that $conc(\nu_\epsilon, \sigma_{G_p})$ substitutes the input symbols $\varsigma^0_{G_p}$ with σ_{G_p}:

Definition 9 (Precision of leaf nodes). Under an analysis state a, a leaf node $\epsilon \in sum_a(m)$ is precise if and only if any global input valuations σ_{G_p} satisfying $int(a, pc_\epsilon)$, $\langle \sigma_{G_p}, f^0_m\rangle \rightarrow^* conc(\nu_\epsilon, \sigma_{G_p})$ i.e. when starting the execution with the input σ_{G_p} and the initial frame for the method m, the execution reaches the concretization of the symbolic state ν_ϵ. The summaries sum_a are precise if any leaf node ϵ of the summary $sum_a(m)$ of any method m is precise.

For progress, it is essential that symbolic interpretation maintains *totality* of the summaries. A reachable concrete state s is on the frontier if all predecessors in the execution to s are concluded reachable, but s is not concluded reachable. The summaries are total if any concrete state s on the frontier is a concretization of some unknown leaf node ϵ in the summary of some method m. Informally, this is a kind of completeness guarantee for symbolic interpretation. For any concrete state on the frontier, the analysis can make the "right" choice. Totality implies that leaf nodes may not be marked unreachable if $Check$ succeeds in the current analysis state. For this reason, the call and return rules need to reconsider some unreachable leaf nodes.

Definition 10 (Frontier). In an analysis state a, a concrete state s is on the frontier (denoted $front(pr, a, s)$) if and only if there exists a reachable concrete state s' such that all states on the execution $s^0_{pr} \rightarrow^* s'$ are reachable, and $s' \rightarrow s$ and $\vdash^a reach(pr, s)$ is false.

Definition 11 (Totality of summaries). In an analysis state a, the summaries sum_a are total if and only if for any concrete state s on the frontier, there exists an unknown leaf node $\epsilon \in sum_a(m)$ in the summary of some method m such that there is a global valuation σ_{G_p} satisfying pc_ϵ and s is a concretization of ν_ϵ.

In addition, symbolic interpretation also maintains precision and totality of the invocations.

Definition 12 (Precision of invocations). Under an analysis state a, an invocation $\langle m_s, m_t, \varsigma_G, pc\rangle \in invs_a$ is precise if and only if for all global input valuations σ_{G_p} , if σ_{G_p} satisfies $int(a, pc)$ then $\langle \sigma_{G_p}, f^0_{m_s}\rangle \rightarrow^* \langle \sigma_{G'_p}, \langle m, n, \sigma_L\rangle\rangle$ where $\lambda_m(n) =$ **call** m_t, n' and $\sigma_{G'_p}$ is the concretization of ς_G with input σ_{G_p}. In other words, when starting the execution with the input σ_{G_p} and the initial frame for the method m_s, the execution reaches an invocation of the method m_t where the global variables are the concretization of ς_G with input σ_{G_p}.

Definition 13 (Totality of invocations). In an analysis state a, the invocations $invs_a$ are total if and only if for any method m and any concrete state s such that $allReach(a, \langle \sigma_{G_p}, f^0_m\rangle \rightarrow^* s)$, if $s = \langle \sigma_{G'_p}, \langle m, n, \sigma_L\rangle\rangle$ and $\lambda_m(n) =$ **call** m_t, n', then there exists an invocation $\langle m, m_t, \varsigma_G, pc\rangle \in invs_a$ such that σ_{G_p} satisfies $int(a, pc)$ and $\sigma_{G'_p}$ is the concretization of ς_G with σ_{G_p}.

6.3. Reachability checking

Finally, to check reachability ($Check(a, m, pc)$), the algorithm globalizes the path condition pc based on the invocation graph inv_a, substitutes the symbols $\varsigma^0_{G_p}$ by their interpretation $interps(sum_a)$, and uses an SMT-solver to check the satisfiability of the resulting constraints. The globalization $glob_p(a, m, pc)$ globalizes the constraint pc in the context of m using the invocation graph $invs_a$ and is defined inductively as follows:

- If $m = m^0_p$ then $glob_p(a, m, pc) = pc$
- If $m \neq m^0_p$ then $glob_p(a, m, pc) =$
  $\bigvee_{\langle m_s, m, \varsigma_G, pc'\rangle \in invs_a} glob_p(a, m_s, pc' \wedge pc[\bigcup_{v \in G_p} \varsigma^0_{G_p}(v) \mapsto \varsigma_{G_p}(v)])$

In the absence of recursion, the invocation graph is cycle free, and the inductive definition is well-founded.

For precision, it is important that $Check(a, m, pc)$ only returns true when there is a reachable state s where the execution enters m and the global variables satisfy pc (*precision of Check*). This follows from precision of the leaf nodes, the precision of the interpretations and the soundness of the SMT-solver as a satisfiability checker.

```
1  class IncDecTest {
2    int IncDec(int N, int n, int nbInc, int nbDec) {
3      if (n < 0 || n > N || nbInc < 0 ||
4        nbInc > N || nbDec < 0 || nbDec > N)
5        return 2;
6      if (Inc(n, nbInc, nbDec)) return 0;
7      else return 1;
8    }
9    bool Inc(int n, int nbInc, int nbDec) {
10     int m = n;
11     for (int i = 0; i < nbInc; i++) m++;
12     return Dec(m, nbDec);
13   }
14   bool Dec(int m, int nbDec) {
15     int k = m;
16     for (int i = 0; i < nbDec; i++) k--;
17     return Check(k);
18   }
19   bool Check(int k) {
20     if (k == 1) return true;
21     else return false;
22   }
23 }
```

Fig. 11. IncDec

Definition 14 (Precision of reachability check). Reachability checking ($Check$) is precise if and only if for each program p, each reachable analysis state a, each method $m \in M_p$ and each condition pc, if $Check(a, m, pc) = true$ then there exists global input valuation σ_{G_p} such that m is invoked with global variable valuation σ_{G_p} and $int(a, pc)[\theta_p^0(\sigma_{G_p})]$ holds. In this definition, $\theta_p^0(\sigma_{G_p})$ is the substitution of the initial global variable symbols by their corresponding valuation in σ_{G_p}.

The contrary is not the case: If there is an execution that enters m where the global variables satisfy pc, $Check(a, m, pc)$ need not return true because this execution might follow an unexplored path through some method. For progress, it is only necessary that $Check(a, m, pc)$ holds if the execution that enters m where the global variables satisfy pc only uses concrete states that are concluded reachable (*Restricted completeness*). This requires completeness of the SMT-solver as a satisfiability checker.

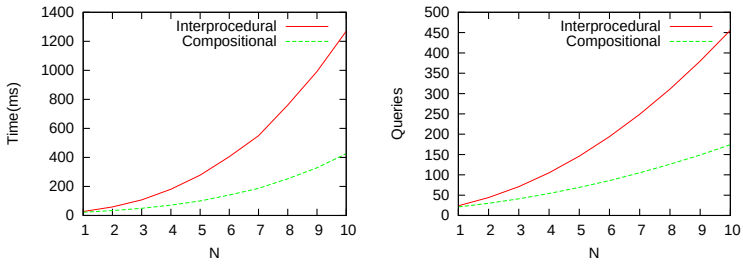
Definition 15 (Restricted-Completeness of reachability check). Reachability checking ($Check$) is restricted-complete if and only if for each program p, each reachable analysis state a, if there exists a concrete state s such that $allReach(a, s_p^0 \rightarrow^* s)$ and s calls m with the global variable valuation σ_{G_p} satisfying the condition pc, then $Check(a, m, pc) = true$.

6.4. Implementation

To show that the algorithm in Section 6 provides similar speedup as other compositional symbolic execution tools [AGT08, God07], we have implemented one instantiation of the framework where the programming language is the intermediate language of the .NET platform [ECM06]. For parsing bytecode, we use the Mono.Cecil [Eva] library and as constraint solver we use Z3 [dMB08].

Our tool is based on dynamic symbolic execution, a variant of symbolic execution where the program is executed with real inputs and monitored during execution to build a symbolic representation on the side. Dynamic symbolic execution prevents false positives because it detects and reports preciseness problems at runtime. Whenever the real execution does not follow the intended path the tool reports it. This was useful to debug the symbolic interpretation rules before we proved precision.

We applied both the compositional and the interprocedural version of our tool on the example *IncDec* described in Figure 11 which is inspired by [AGT08]. In this example, an initial number n is first increased by *nbInc*, then decreased by *nbDec* and finally check for equality with the target value 1. However, the addition and subtraction operations are encoded using loops to complicate the analysis. Initially, all values



(a) Average time (b) Average queries

Fig. 12. Results of experiments

are bounded between 0 and N, therefore we can run the analysis until all paths through the program are explored.

We measured the execution time (Figure 12a) and the amount of queries (Figure 12b) in function of the bound N. All experiments were conducted on an Intel Core 2 Duo T7500 (2.2 GHz) with 4Gb of memory. We repeated each experiment 10 times and report the average results of all experiments.

Figure 12b shows that the amount of queries performed by the interprocedural tool is quadratic in the bound N. The compositional tool clearly does less queries and is almost linear in N. It is not entirely linear due to re-evaluating unreachable leaf nodes. In Figure 12a, one can see that the execution time is clearly less for the compositional tool. Although the compositional queries are more expensive than the interprocedural queries, the sheer number of queries causes the interprocedural tool to take more time.

6.5. Recursion

In Section 4, we introduced the simplifying assumption that there are no (mutually) recursive methods. This assumption only marginally limits the expressiveness of the formal model since loops already introduce similar complexity. However, the proof depend on this simplification because they require the invocation graph to be cycle-free.

In our implementation, we maintain this property by using the traditional symbolic interpretation rules whenever the compositional rules may cause a cycle in the invocation graph. To achieve this, we infer a rank, a strict total order over all methods, and only use the compositional rules when the rank strictly decreases. As a result, recursion is unrolled incrementally in a similar fashion as loops.

6.6. Discussion

Previous work [God07, MS07, AGT08] has shown that compositional symbolic execution is a promising approach to improve the scalability of symbolic execution. In Section 6.4, we have also shown an example where compositional symbolic execution clearly performs less queries and therefore scales better than symbolic execution. However, compositional symbolic execution may not always work better.

For example, while compositional symbolic execution reduces the number of constraint solver queries, each query may contain a larger number of disjunctions and may therefore be harder for the constraint solver. In some sense, standard symbolic execution eagerly case splits at each branch and aggressively simplifies the constraints based on these decisions. In addition, symbolic execution may use specialized heuristics to decide the next branch where to perform case analysis. On the other hand, the constraint solver is designed to handle all constraints and uses general purpose heuristics which may not work optimally.

Alternatively, compositional symbolic execution may not always reduce the number of queries. While the call rule for standard symbolic execution (See Figure 6) is fully deterministic, compositional symbolic execution requires the guarantee that the method has already been explored for the given arguments before using the summary. Therefore, the call rule for compositional symbolic execution (See Figure 13) performs a case split. To deal with exceptions, this call rule requires another case split to determine whether the call

returns normally or with an exception. Therefore, compositional symbolic execution may also lead to a larger number of queries. While most of these queries are simple, they may have an averse effect on scalability.

7. Properties

In this section, we show that the algorithm of Section 6 is precise, and we show that it is also progressing as long as the choices are fair.

In Section 6, we defined the key invariants that enable precision and progress. First, we show that the algorithm maintains these properties. We say an analysis state is valid if it satisfies its invariants:

Definition 16 (Validity of analysis state). An analysis state a is precise if and only if

1. Each leaf node ϵ of the summary $sum_a(m)$ of each method m is precise.
2. Each finished leaf node ϵ of the summary $sum_a(m)$ of each method m is returning from m.
3. The invocation graph $invs_a$ is cycle free.
4. All invocations $inv \in invs_a$ are precise.
5. Each leaf node ϵ in rs_a is precise, and globally reachable: there exists a global variable valuation σ_{G_p} such that m is invoked with valuation σ_{G_p} that satisfies $int(a, pc)$ (where m is active method of the symbolic state of ϵ).
6. The summaries sum_a are total.
7. The invocation graph $invs_a$ is total.
8. For each all-reachable concrete state s that returns from m, $sum_a(m)$ contains a finished node ϵ such that s is a concretization of ν_ϵ.

Lemma 1 (Precision of interpretations). For any program p and valid analysis state a, the interpretations are precise.

Proof. For any valid analysis state, the invocation graph is cycle free. Therefore, we can do induction on the depth of the invocation graph.

- If the method m does not invoke another method, the proof is straightforward from the definition of the interpretations and the precision of the summary of m.
- Otherwise, the methods that can be called have a smaller depth in the invocation graph. The induction hypothesis implies that the interpretations of those methods are correct. The interpretation $int(a, pc)$ of a constraint pc substitutes the uninterpreted function symbols from these methods with their interpretation. Using the definition of the interpretations and the validity of the summary of m, we can again prove that the interpretation is precise.

□

Lemma 2 (Precision of reachability checking). For any program p and valid analysis state a, the reachability checking is precise.

Proof. By induction on the depth of the invocation graph.

- Base case: depth is zero, and therefore $m = m_p^0$. Since the interpretations are precise, everything depends on the ability of the SMT-solver to find a satisfying assignment. Therefore, if the SMT-solver is sound as a constraint solver, i.e. the solver only returns true if there actually exists a solution, then the lemma holds.
- Using the induction hypothesis and the precision of the invocations.

□

Lemma 3 (Restricted completeness of interpretations). For any program p and valid analysis state a, the interpretations are restricted complete.

Proof. By induction on the depth of the invocation graph.

- Depth is zero, and therefore $m = m_p^0$. By validity, there is a finished leaf node $\epsilon \in sum_a(m)$ for each all-reachable concrete state s that returns from m such that s is a concretization of ν_ϵ. By the definition of *interps*, rc_m returns true, and $rv_{m,v}$ equals the concrete value for the global variables in s.
- Using the induction hypothesis and totality of the invocations.

□

Lemma 4 (Restricted completeness of reachability checking). For any program p and valid analysis state a, reachability checking is restricted complete.

Proof. By induction on the depth of the invocation graph.

- Depth is zero, and therefore $m = m_p^0$. Using totality of the summaries, and restricted completeness of the interpretations. Therefore, if the smt-solver is complete (as a constraint solver, i.e. if there is a satisfying solution then the solver returns true) then the lemma holds
- Using the induction hypothesis and restricted completeness of the interpretations.

□

Lemma 5 (Validity of the initial analysis state). For any program p, the initial analysis state a_p^0 is valid.

Proof. Follows immediately from the definition of the initial analysis state. □

Lemma 6 (Maintenance of validity). For any program p, and analysis states a, a', if a is valid and $a \Rightarrow a'$, then a' is valid.

Proof. First, the method CHOOSE chooses a method m and an unknown node $\epsilon = \langle unk, \nu, pc \rangle \in sum_a(m)$. If no such node can be found in any method, the analysis is completed. There is no a' such that $a \Rightarrow a'$, so the theorem trivially holds.

If a method and unknown node has been selected, analysis continues with reachability checking. Suppose $Check(a, m, pc_\epsilon) = false$, then there is no global input valuation σ_{G_p} that satisfies $int(a, pc)[\theta_p^0(\sigma_{G_p})]$ and reaches m. The algorithm replaces ϵ with an unreachable node. This unreachable node is valid since the unknown node was valid, and the path condition and execution depth remain the same. In addition, it is impossible to violate totality of the summaries since check is restricted complete.

If $Check(a, m, pc_\epsilon) = true$ then there exists a global input valuation σ_{G_p} satisfying $int(a, pc)$ such that m is globally reachable. The algorithm refines the leaf node and executes one or more steps of symbolic interpretation. In what follows, we show that one step of symbolic interpretation results in a new, valid analysis state. The same result for multiple steps can be obtained by induction on the number of steps. The added reachability element is also precise (by the precision of ν_ϵ) and globally reachable (by the precision of the reachability check).

The remainder of the proof is a case split on the symbolic interpretation rule:

assign The assign rule only updates the symbolic state, and therefore the members of the leaf nodes remain the same. Since ν_ϵ is precise, all valuations lead to a concrete state s where the projected execution rule ASSIGN applies. The symbolic interpretation rule applies the same mutation to the symbolic state, and is therefore again precise. In addition, if sum_a was total then the new summaries are also total, since all members of ϵ are now member of the new leaf node.

cond The argument for conditional branches is similar as for assignment except that there are now two new nodes ϵ_1 and ϵ_2. By precision of ϵ, all executions lead to a concrete state s where either COND-T or COND-F applies. When COND-T applies, the input valuation becomes a member of ϵ_1 and ϵ_1 is again precise. Alternatively, when COND-F applies, the input valuation becomes a member of ϵ_2 and ϵ_2 is again precise. In addition, if sum_a was total then the new summaries are also total, since all members of ϵ are now member of ϵ_1 or ϵ_2.

call The call rule creates a new leaf node ϵ' that represents the symbolic state after returning from the target method. Since ϵ was precise, and the interpretations are precise, ϵ' is also precise.
In addition, the new invocation is added to the invocation graph and the function *rec* re-evaluates unreachable nodes. Marking unreachable nodes as unknown does not affect the precision of the nodes. Since all unreachable nodes of all methods that are reachable in the invocation graph are re-evaluated,

the new summaries are total again. All inputs that previously were members of ϵ are now either reachable, or a member of some unknown node.
Because the program is not (mutually) recursive, the new invocation does not introduce cycles. In addition, by precision of ϵ, the new invocation is also precise.

ret The ret rule marks the unknown node as finished. The members of the node remain the same, and the symbolic state remains precise.
In addition, RET re-evaluates all unreachable nodes that depend on the return of m. All unreachable nodes that can become reachable are re-evaluated. Therefore, the new summaries are total again.

□

Corollary 1 (All reachable analysis state are valid). For any program pr, and any analysis state a such that $a^0_{pr} \Rightarrow^* a$, a is valid.

Precision follows immediately from validity, since each leaf node in the reachability set is precise and *Check* succeeds.

Theorem 7.1 (Precision). The algorithm is precise.

The argument for progress is more complicated. First, we show that compositional symbolic execution is monotonous.

Theorem 7.2 (Monotonicity). For each program pr, concrete state s, and analysis states a, a' such that $a \Rightarrow a'$, if $\vdash^a reach(pr, s)$ then $\vdash^{a'} reach(pr, s)$.

Proof. Follows from the fact that (a) the relation $\Rightarrow$ never removes reachable leaf nodes ($rs_a \subseteq rs_{a'}$), (b) the relation $\Rightarrow$ never removes invocations ($invs_a \subseteq invs_{a'}$). □

Since the search tree of the algorithm is potentially infinite, monotonicity is not sufficient to find all reachable states: The algorithm might get stuck exploring only a subspace of the program. Fortunately, this can not happen if the analysis is *fair*, i.e. if each unknown node is eventually chosen by the algorithm.

Definition 17 (Fairness). An application strategy of the compositional symbolic execution algorithm is *fair* if and only if for any analysis state a such that $a^0_{pr} \Rightarrow^* a$, for any unknown node $\epsilon \in sum_a(m)$, the algorithm always eventually chooses $\langle m, \epsilon \rangle$.

Finally, we show that compositional symbolic execution algorithm is progressing if it is fair. The proof shows a slightly stronger property, namely that there always eventually is an analysis state where all concrete states on the execution trace that reaches s are concluded reachable. This is essential since it gives a stronger induction hypothesis: we assume that all but the last concrete state s is concluded reachable and we show that the analysis always eventually reaches an analysis state where s is also concluded reachable. This hypothesis is necessary since a state might only be reachable from one invocation that has not yet been discovered, whereas its predecessor is already reachable based on another invocation. Together with totality of the summaries and restricted completeness of reachability checking, this allows a compact and intuitive proof for progress.

Theorem 7.3 (Progress). If the compositional symbolic execution algorithm is fair, then it is progressing.

Proof. By induction on $\rightarrow^*$.

Base step If s is the initial state, then $\vdash^a reach(pr, s)$ holds after applying the only possible analysis step.

Induction step If $s^0_{pr} \rightarrow^* s'$ and $s' \rightarrow^* s$ and there always eventually is a reachable analysis state a' such that all concrete states from s^0_{pr} to s' are concluded reachable in a', then we must show that there always eventually is a reachable analysis state a such that $\vdash^a reach(pr, s)$. If $\vdash^{a'} reach(pr, s)$ already holds, then the proof is trivial.

1. First, we show that there exists an unknown node $\epsilon \in sum_{a'}(m)$ such that $Check(a', m, pc_\epsilon) = true$ and s is a concretization of ϵ. This means that if we choose $\langle m, \epsilon \rangle$, then the state s will become reachable in the next analysis state. This follows from the fact that the summaries are total, and restricted completeness of check.

2. By fairness, there always eventually exists a reachable analysis state a'' such that $\langle m, \epsilon \rangle$ has not been chosen yet, and is chosen in the next analysis step. Since $\langle m, \epsilon \rangle$ has not been chosen, it must still be in the summary of m ($\epsilon \in sum_{a''}(m)$). Because invocations are never removed ($invs_a \subseteq invs_{a''}$), the method $Check$ is monotonous and $Check(a'', m, pc_\epsilon) = true$. Therefore, if $\langle m, \epsilon \rangle$ is chosen in $a'' \Rightarrow a$, then $\vdash^a reach(pr, s)$.

□

8. Related work

Compositional symbolic execution was first introduced in the context of SMART [God07], as an extension of the automatic testing tool DART [GKS05]. The authors informally argue that SMART is sound and complete (as a bugfinder) relatively to DART. In addition, DART is always sound (precise) and it is complete when it terminates [GKS05]. The precision proofs depends critically on the dynamic aspect of SMART and DART. The proof in this paper only depends on the precision of the interpretation rules. When the interpretation rules are imprecise in SMART or DART, it either causes incompleteness or non-termination. In addition, the progress property is stronger than completeness upon termination.

The principle of lazy expansion used in LATEST [MS07] is highly related to compositional symbolic execution. However, since LATEST only explores the top-level method, it can optimize the construction of summaries by focusing on relevant paths for the top-level method. The authors claim soundness and completeness relative to CUTE[SMA05] for programs with a finite computation tree. Precision and progress are stronger, since they are also defined for programs without finite computation tree.

With demand-driven compositional symbolic execution [AGT08], the dependency on the inner-most first search order of SMART is lifted. To achieve this, function summaries are encoded in the SMT-solver. In addition, the algorithm allows the SMT solver to construct inputs that follow unexplored paths through some methods. Such inputs may not reach their actual target but they always explore some new part of the program. This may be useful to alleviate the imperfection of SMT solvers. We did not incorporate this in our framework, but the results can easily be extended. The authors claim relative completeness (as a bugfinder), and termination for programs with finite amounts of paths. The progress property in this paper is less algorithm specific and therefore more clear. In addition, it lifts the need for a termination argument. In the absence of fairness, demand-driven compositional symbolic execution does not satisfy the stronger progress property.

Finally, the system SMASH [GNRT10] combines the aspect of compositional analysis with may-must alternation. SMASH significantly outperforms both may-only, must-only and non-compositional may-must analysis. The analysis in this paper is a must analysis. As part of the soundness argument, the authors show that the must analysis of SMASH is precise. In addition, they show that the may analysis of SMASH is sound. Unfortunately, the combination of a sound may analysis with a precise must analysis is not necessarily a semi-decision procedure.

The idea of compositional program analysis is highly related to the concept of modular verification. With modular verification, each module can be separately verified with respect to its specification and the composition is guaranteed to be sound. The KeYtool [ABB+05] and KIV [Rei92] are using compositional symbolic execution to perform modular verification. Unlike compositional program analysis, these systems require human interaction to provide the necessary specifications for each module. In addition, these systems focus on soundness rather than precision. In this context, detailed formal models have been created to study the properties of compositional symbolic execution [BHS07, Pla04]. Compositional symbolic execution has also been combined with abstract interpretation to automatically infer loop invariants [Wei09, BHW09].

Beckert et. al. [BG07] use (compositional) symbolic execution to generate partial method contracts which can be used to generate tests.

The COSTA/PET [AGZP10, GzAP10] system is an alternative approach to do symbolic execution for test case generation. In this system, the program is transformed into a Constraint Logic Program [MS98] which is then partially evaluated into a test case generator.

9. Conclusion

In this paper, we have created a formal framework based on transition systems to analyze the fundamental properties of reachability analysis algorithms. We use this framework to model a symbolic execution algorithm and a compositional symbolic execution algorithm for a small but powerful calculus. In addition, we have proven that the compositional algorithm is precise, and makes progress if the choices are fair. Finally, we have shown preliminary results of an implementation of the compositional algorithm that is precise and progressing, and hence is a semi-decision procedure.

References

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[AGT08] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, pages 367–381, 2008.

[AGZP10] Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. Pet: a partial evaluation-based test case generation tool for java bytecode. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 25–28, New York, NY, USA, 2010. ACM.

[APV07] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Jpf-se: A symbolic execution extension to Java Pathfinder. In *Proc. of TACAS 2007*, pages 134–138, Braga, Portugal, March 2007.

[BG07] Bernhard Beckert and Christoph Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In *Proceedings, Testing and Proofs*. Springer, 2007.

[BHK+07] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Dawn Song. BitScope: Automatically dissecting malicious binaries. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University, March 2007.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[BHW09] Richard Bubel, Reiner Hähnle, and Benjamin WeiSS. Abstract interpretation of symbolic execution with explicit state updates. In Frank de Boer, Marcello Bonsangue, and Eric Madelaine, editors, *Formal Methods for Components and Objects*, volume 5751 of *Lecture Notes in Computer Science*, pages 247–277. Springer Berlin / Heidelberg, 2009.

[CCC+05] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. *SIGOPS Oper. Syst. Rev.*, 39(5):133–147, 2005.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *Proc. of CCS '06*, pages 322–335, 2006.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[ECM06] ECMA International. *Standard ECMA-335: Common Language Infrastructure*, 4th edition edition, June 2006.

[Eva] Jb Evain. Cecil. http://www.mono-project.com/Cecil.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. *SIGPLAN Notices*, 40(6):213–223, 2005.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of NDSS'08 (Network and Distributed Systems Security)*, 2008.

[GNRT10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. *SIGPLAN Not.*, 45(1):43–56, 2010.

[God07] Patrice Godefroid. Compositional dynamic test generation. In *Proc. of POPL '07*, pages 47–54, 2007.

[GzAP10] Miguel GÔmez-zamalloa, Elvira Albert, and GermÂn Puebla. Test case generation for object-oriented imperative languages in clp*. *Theory Pract. Log. Program.*, 10(4-6):659–674, July 2010.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, March 1998.

[MS07] Rupak Majumdar and Koushik Sen. Latest : Lazy dynamic test input generation. Technical Report UCB/EECS-2007-36, EECS Department, University of California, Berkeley, Mar 2007.

[MW07] David A Molnar and David Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical Report 2007-23, University of California Berkeley, February 2007.

[NRTT09] Aditya V. Nori, Sriram K. Rajamani, Saideep Tetali, and Aditya V. Thakur. The yogi project: Software property checking via static analysis and testing. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 178–181, Berlin, Heidelberg, 2009. Springer-Verlag.

[PDEP08] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proc. of SIGSOFT '08/FSE-16*, 2008.

[Pla04] André Platzer. An object-oriented dynamic logic with updates. Technical report, 2004.

[Rei92] Wolfgang Reif. The kiv system: Systematic construction of verified software. In Deepak Kapur, editor, *Automated DeductionÛCADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 753–757. Springer Berlin / Heidelberg, 1992.

$$\frac{\lambda_m(n) = \textbf{assign } x, e, n' \qquad \textbf{let } \varsigma_T := (\varsigma_G \cup \varsigma_L) \oplus x \mapsto seval(\varsigma_G \cup \varsigma_L, e)}{a \| m, \langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle \Rightarrow a \| m, \{\langle unk, \langle \varsigma_G, \langle m, n', \varsigma_T|_{L_m} \rangle \rangle, pc \rangle\}} \text{ ASSIGN}$$

$$\frac{\lambda_m(n) = \textbf{if } e, n_t, n_f \qquad \textbf{let } b := seval(\varsigma_G \cup \varsigma_L, e) \qquad \textbf{let } \epsilon_1 := \langle unk, \langle \varsigma_G, \langle m, n_t, \varsigma_L \rangle \rangle, pc \wedge b \rangle \qquad \textbf{let } \epsilon_2 := \langle unk, \langle \varsigma_G, \langle m, n_f, \varsigma_L \rangle \rangle, pc \wedge \neg b \rangle}{a \| m, \langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle \Rightarrow a \| m, \{\epsilon_1, \epsilon_2\}} \text{ COND}$$

$$\frac{\lambda_m(n) = \textbf{call } m_t, n' \qquad \textbf{let } \varsigma'_G := \bigcup_{v \in G_p} v \mapsto rv_{m,v}(\varsigma_G) \qquad \textbf{let } \epsilon := \langle unk, \langle \varsigma'_G, \langle m, n', \varsigma_L \rangle \rangle, pc \wedge \ rc_m(\varsigma_G) \rangle \qquad \textbf{let } invs'_a := invs_a \cup \{\langle m, m_t, \varsigma_G, pc \rangle\} \qquad \textbf{let } a' := rec(\langle sum_a, invs'_a, rs_a \rangle, m_t)}{a \| m, \langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle \Rightarrow a' \| m, \{\epsilon\}} \text{ CALL}$$

$$\frac{\lambda_m(n) = \textbf{ret} \qquad \textbf{let } a' := rer(a, m)}{a \| m, \langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle \Rightarrow a' \| m, \{\langle fin, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle\}} \text{ RET}$$

Fig. 13. Interpretation rules

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proc. of ESEC/FSE'05*, pages 263–272, New York, NY, USA, 2005. ACM Press.

[TdH08] Nikolai Tillmann and Jonathan de Halleux. Pex - white box test generation for .net. In *Proc. of Tests and Proofs '08*, pages 134–153. Springer Berlin / Heidelberg, 2008.

[TS05] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 253–262, New York, NY, USA, 2005. ACM.

[VP11] Dries Vanoverberghe and Frank Piessens. Theoretical aspects of compositional symbolic execution. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Proceedings of Fundamental Approaches to Software Engineering (FASE 2011)*, volume 6603 of *Lecture Notes in Computer Science*, pages 247–261. Springer Berlin / Heidelberg, 2011.

[Wei09] Benjamin Weiß. Predicate abstraction in a program logic calculus. In *Proceedings of the 7th International Conference on Integrated Formal Methods*, IFM '09, pages 136–150, Berlin, Heidelberg, 2009. Springer-Verlag.

A. Symbolic interpretation rules

Figure 13 shows the symbolic interpretation rules for the language of Section 4. Each rule is structured as follows:

$$\frac{\lambda_m(n) = \ldots}{a \| m, \langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle \Rightarrow a' \| m, \pi}$$

When a rule is applied to an analysis state a with a chosen method m and leaf node $\langle unk, \langle \varsigma_G, \langle m, n, \varsigma_L \rangle \rangle, pc \rangle$, the rule potentially changes the analysis state to a' and returns a new set of leaf nodes π.

In the remainder of this section, we define the functions $rec(a, m)$ an $rer(a, m)$ which are used by the rules for call and return.

The function $rec(a, m)$ (Re-Evaluate Call) returns a new analysis state where all unreachable nodes in each method which is reachable from m in the invocation graph are changed into unknown nodes. A method m' is reachable from a method m in a set of invocations $invs$ (denoted $reachM(invs, m, m')$) if and only if $m = m'$ or there exists an invocation $\langle m, m'', \varsigma_G, pc \rangle \in invs$ such that m' is reachable from m'' in $invs$. The set of reachable methods from m in a set of invocations $invs$ is denoted $rms(invs, m) = \{m' \in M_p | reachM(invs, m, m')\}$.

Formally:

$$rec(a, m) = \langle sum_a \oplus \bigcup_{m' \in rms(invs_a, m)} m' \mapsto rep(sum_a(m)), invs_a, rs_a \rangle$$

where $rep(\pi) = \{\epsilon | \epsilon \in \pi, state_\epsilon \neq unr\} \cup \{\langle unk, \nu_\epsilon, pc_\epsilon \rangle | \epsilon \in \pi, state_\epsilon = unr\}$ replaces all unreachable nodes by unknown nodes in the partition π.

The function $rer(a, m)$ (Re-Evaluate Return) returns a new analysis state where all unreachable nodes that depend on the return of the method m are changed into unknown nodes. A node depends on the return of m if its path condition depends on the return condition of m, or because it is reachable through an invocation where the path condition depends on the return condition of m.

A symbolic constraint pc depends on the return of a method m in the invocation graph $invs$ (denoted $dep(pc, m, invs)$) if and only if there exists an invocation $\langle m', m, \varsigma_G, pc \rangle \in invs$ such that pc contains the subterm $rc_m(\varsigma_G)$. A node ϵ or an invocation inv depends on the return of a method m in the invocation graph $invs$ (also denoted $dep(\epsilon, m, invs)$ and $dep(inv, m, invs)$) if and only if their symbolic constraint pc_ϵ or pc_{inv} depends on m in $invs$. The set $rmti(invs, m)$ of reachable methods through an invocation that depends on m in the invocation graph $invs$) is defined by $\{m' | m' \in M_p, inv \in invs, dep(inv, m, invs), reachM(invs, inv_{m_t}, m')\}$. Then $rer(a, m) = rer_2(rer_1(a, m), m)$ consists of two phases:

$$rer_1(a, m) = \langle sum_a \oplus \bigcup_{i \in calls(invs_a, m)} m_{s_i} \mapsto repd(sum_a(m_{s_i}), m_{s_i}), invs_a, rs_a \rangle$$

where $calls(invs, m)$ is the set of invocations to the method m in the current invocation graph $invs$, i.e. $calls(invs, m) = \{i | i \in invs, m_{s_i} = m\}$ and $repd(\pi, m, invs) = \{\epsilon | \epsilon \in \pi, state_\epsilon \neq unr \vee \neg dep(\epsilon, m, invs)\} \cup \{\langle unk, \nu_\epsilon, pc_\epsilon \rangle | \epsilon \in \pi, state_\epsilon = unr, dep(\epsilon, m, invs)\}$ re-evaluates the nodes of π that depend on the return of m.

$$rer_2(a, m) = \langle sum_a \oplus \bigcup_{m' \in rmti(invs_a, m)} m' \mapsto rep(sum_a(m), inv), invs_a, rs_a \rangle$$